Profile-Guided Field Externalization in an Ahead-of-Time Compiler

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

Abstract

Field externalization is a technique to reduce the footprint of objects by removing fields that most frequently contain zero or null. While researchers have developed ways to bring this optimization into the Java world, these have been limited to research compilers or virtual machines for embedded systems. In this work, we present a novel field externalization technique that uses information from static analysis and profiling to determine externalizable fields. During compilation, we remove those fields and define *companion classes*. These are used in case of non-default-value writes to the externalized fields. Our approach also correctly handles synchronization to prevent issues in multithreaded environments. We integrated our approach into the modern Java ahead-of-time compiler GraalVM Native Image. We conducted an evaluation on a diverse set of benchmarks that includes standard and microservice-based benchmarks. For standard benchmarks, our approach reduces the total allocated bytes by 2.76% and the maximum resident set size by 2.55%. For microservice benchmarks, the allocated bytes could be reduced by 6.88% and the maximum resident set size by 2.45%.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Object oriented languages; Software and its engineering → Classes and objects

Keywords and phrases compilation, instrumentation, profiling, fields, externalization, memory footprint reduction, memory footprint optimization

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Funding Anonymous funding

Acknowledgements Anonymous acknowledgements

1 Introduction

Optimizing a program for speed, efficiency, and safety is a crucial goal of most compilers. In modern languages, such optimizations frequently concern objects and similar structured types. Java is a language that enables object-oriented programming at a high abstraction level without sacrificing performance. As manyfold as the usage scenarios of objects are in Java, as varied are the optimizations that compilers can apply. Optimizations on objects tend to fall into two categories: Reducing the memory footprint of objects and improving the efficiency of accesses to the objects' properties and methods. Object allocations are made more efficient by directly allocating into thread-local allocation buffers [23] or are eliminated altogether via escape analysis [11, 52]. Escape analysis typically also removes field accesses

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:31
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
class Order {
    long orderId;
    Item[] items;
    double shippingCosts;
    String discountCode;
}
```

externalizable

(a) `Order` class with an ID, an array of (unspecified) items, shipping costs, and a discount code. The example assumes that `shippingCosts` and `discountCode` are rarely needed and thus externalizable.

```
class Order {
    long orderId;
    Item[] items;
    Order$Companion _ref;
}

class Order$Companion {
    double shippingCosts;
    String discountCode;
}
```

(b) Externalized version of `Order` including its generated companion class that holds the externalized fields. The original class is augmented with a new `_ref` field that references the companion object if allocated. In later parts of the paper we detail improvements that allow us to eliminate this field.

Figure 1 Original and externalized representation of an `Order` class that is used as a continuous example to explain various aspects of our field externalization implementation.

via scalar replacement, thus improving performance. Dynamic dispatch on methods is also a frequent target for optimizations. Compilers try to devirtualize calls [22], inline them [1, 26], or use inline caches [10, 12, 18, 19, 26, 64] to optimize for the most common cases. Other optimizations aim to shrink the object header [24, 58], a prefix in every object that references, among other things, type information and the virtual method table. While compilers also try to minimize the memory consumption of objects, the range of techniques [8, 56] applicable in this area has been more limited so far.

When analyzing heap dumps of Java applications, we noticed that certain fields of objects often just hold their default values. Other researchers have pointed out similar findings [3, 9, 16, 47]. This shows potential for optimizations, as such objects occupy more memory than necessary. Especially in cloud and serverless computing, reducing the memory footprint of applications is important. Thus, we propose a novel form of *field externalization*—a technique for removing such fields, thereby reducing the memory footprint of objects and thus the overall memory consumption of an application.

The `Order` class shown in Figure 1a is used as a running example throughout this paper. In this example, an order consists of an ID, an array of order items (the implementation of which is irrelevant for this example), the shipping costs, and a discount code. We assume that the first two fields are non-zero and non-null in most objects. However, both `shippingCosts` and `discountCode` are not; they are mostly zero or null (the default values for these fields) and only hold relevant values in a small number of objects.

Via field externalization, we can optimize the layout of the `Order` class by removing fields that hold default values in most cases. This results in the layout presented on the left-hand side of Figure 1b. However, as optimizations have to ensure that also corner cases are handled correctly, we must be able to handle `shippingCosts` and `discountCode` if they are ever set to a non-default value. Therefore, we create a new class, the so-called *companion type* `Order$Companion`, that stores the *externalized fields*, i.e., the fields that were removed from the original `Order` class. Additionally, we introduce a pointer in the original class—we call this the *companion reference field*. Initially, this pointer is null. If a non-default value is assigned to an externalized field, an instance of the companion type—the *companion object*—is allocated to store the field value. This companion object is assigned to the companion reference field. Hence, writing a non-default value to an externalized field introduces some overhead. However, as allocating the companion object should only be necessary in a few cases, these costs are outweighed by the reduced overall footprint of the other objects.

Various researchers have developed approaches to integrate field externalization into compilers [3, 9, 16, 47]. Notably, most are based on just-in-time (JIT) compilers [3, 9, 16], compilers for embedded systems [9, 16, 47], or research compilers [3, 47]. These approaches have shown promising results in combating memory consumption in Java programs. However, we could not find any contemporary approach that reliably enables field externalization in a production-grade compiler without compromising language or run-time features such as multithreading.

Therefore, we propose a novel approach for field externalization in the state-of-the-art ahead-of-time (AOT) compiler framework GraalVM Native Image [61, 62]. It works by first gathering profiling information for a target program in an offline profiling run. Then, we combine the profiling information with information from Native Image's points-to analysis [17, 46, 51] to identify and subsequently externalize fields that most often hold their default value. Furthermore, we developed a synchronization mechanism for accesses to externalized fields that adheres to the Java Memory Model [30].

With this work, we make the following contributions:

1. A profiling approach that utilizes both static analysis as well as field-level profiling to identify fields for optimization.
2. A novel profile-guided field externalization approach in a modern AOT compiler that preserves program semantics even in multithreaded environments.
3. An evaluation of our approach on standard and microservice benchmarks measuring memory footprint, run-time performance, and image size.

The paper is structured as follows: Section 2 describes GraalVM and GraalVM Native Image, focusing especially on components that are important for our approach. In Section 3, we summarize our profiling methodology and its implementation. We explain the detailed approach of field externalization in Section 4, where we also cover notable improvements and details about the integration into Native Image. Section 5 presents the results of applying our approach to a large set of benchmarks. We explain our evaluation methodology and discuss the results in detail. In Section 6, we list the limitations of our approach. In Section 7, we compare our techniques to related work.

2 Background

We integrated our approach into GraalVM Native Image [34, 61, 62] based on GraalVM 23.1 with Java version 21: We use its profile-guided optimization feature [4, 6, 39, 49, 61] to first collect information about field values in a profiling run on an instrumented executable and subsequently externalize rarely written fields based on this profiling information during the compilation of the final executable.

2.1 GraalVM

GraalVM is a high-performance, polyglot virtual machine [34, 64] designed to run programs written in a wide range of languages. It can handle traditional Java-bytecode languages, such as Java, Kotlin, and Scala, as well as languages like JavaScript [35], Ruby [38], Python [33], and even C/C++[43]. The GraalVM compiler is a state-of-the-art JIT compiler within GraalVM [13, 28, 52]. It uses a graph-based intermediate representation (the Graal IR) [13, 14] and performs optimizations such as inlining, constant folding, loop unrolling, and escape analysis [27, 64]. When used as a JIT compiler in a regular Java Virtual Machine (JVM), GraalVM collects run-time information to guide its optimizations. For instance, it identifies hot methods, frequently taken branches, and targets of virtual calls. By exploiting this data, the compiler can further optimize the application according to the observed workload. However, relying on run-time profiling data means that optimizations only take full effect once the application has been running long enough to gather representative insights. This can lead to slower startup times.

2.2 GraalVM Native Image

To provide fast startup times and reduced resource usage, GraalVM offers Native Image [61, 62], an AOT compiler that compiles a Java application ahead of time into a self-contained native binary (the so-called *image*). This binary bundles the application code, all necessary libraries, and a lightweight runtime called *SubstrateVM*, which provides essential services such as threading and garbage collection, into a single executable [36]. Once compiled, no standard JVM is needed at run time, and the startup performance can be improved by up to two orders of magnitude compared to the Java HotSpot VM [62], which can be critical for short-lived processes, serverless functions, or microservices. A key concept in GraalVM Native Image is the closed-world assumption [62]. Under this assumption, all classes, methods, and fields that may possibly be accessed at run time must be known at compile time. This contrasts with the traditional JVM, where classes can be dynamically loaded at run time. The closed-world assumption allows GraalVM Native Image to perform powerful whole-program static analyses.

2.2.1 Points-to Analysis

One important step performed by GraalVM Native Image during static analysis is the points-to analysis [17, 46, 51]. This analysis determines which objects, fields, and methods can be reached from the program's entry point. By understanding which objects point to which other objects (and thus which fields and methods are relevant), Native Image can identify what parts of the code are actually needed. Points-to analysis builds a global, static view of the application's data flow. The analysis can reveal, for example, that a certain type's fields are never written to or that certain methods are never invoked. Since Native Image has all code available at compile time, it uses this analysis to optimize the final binary.

2.2.2 Image Heap

In the context of GraalVM Native Image, the *image heap* refers to a specialized, pre-initialized memory region that is embedded into the native executable [62]. During the Native Image compilation process, GraalVM analyzes the application to determine which objects that are needed at run time can already be created at compile time and stored in the image heap. For example, `java.lang.Class` objects that represent the type descriptors of objects in Java are stored there. Furthermore, class initializers can often be already executed at compile time. Since many objects are already present in the image heap, the native executable doesn't need to perform extensive object allocation and initialization at startup. This leads to significantly reduced startup times compared to traditional JVM applications that rely on JIT compilation and need to allocate all objects at run time [62].

2.2.3 Profile-Guided Optimization

Unlike a JIT compiler, which relies on program behavior observed at run time, an AOT compiler has no direct knowledge of how code executes under real workloads. Without further input, it must rely solely on static heuristics to guide optimizations, which can limit the potential improvements. Profile-guided optimization (PGO) addresses this limitation [4, 6, 39, 49, 61]. PGO is a technique in which the AOT compiler is supplied with run-time execution data collected from a representative workload. This approach mirrors how a JIT compiler uses run-time profiling, but shifts the process to compile time via a two-stage compilation workflow:

1. **Instrumentation and Profiling Phase:**
   First, an instrumented version of the application is generated by Native Image. This instrumented binary contains additional logic to record information about the program as it runs. When executed with a suitable workload that reflects real usage scenarios, the instrumented binary collects detailed run-time data, including call frequencies, branch probabilities, and type occurrences. At the end of this run, the collected profiling information is stored in a file.
2. **Optimized Compilation Phase:**
   In the second stage, the profiling data is fed back into Native Image to produce a new, optimized binary. Equipped with the recorded execution patterns, the AOT compiler can apply more informed optimizations, such as refining method inlining decisions and optimizing frequently executed hot paths.

This two-phase compilation adds complexity and overhead to the build process. However, once the optimized binary is produced, it can be deployed and run without incurring any additional overhead.

PGO enables AOT compilers to tailor optimizations to specific workloads. When the workload employed during the instrumentation phase closely resembles the production environment, the resulting binary is finely tuned for that particular scenario. In essence, PGO provides workload-specific optimizations comparable to those offered by JIT compilers, but in a context where all code is precompiled to native form. This technique is also utilized in related work [7, 15, 25, 29, 57, 63, 65].

It is important to distinguish PGO from machine-learning-based optimization techniques [45], which typically utilize a larger and more diverse set of inputs to optimize a broader range of applications. The objective of machine learning is to train models that can reliably predict optimization opportunities in *new programs* based on extensive training datasets. In contrast to that, PGO aims to optimize a program for a *specific workload*. Developing a generalized version of the program that performs efficiently across different workloads is an explicit non-goal of PGO. Thus, our evaluation is conducted accordingly. We employ the same workload for both profiling and measurement phases, although the profiling workload is smaller in size compared to that used for measurements.

3 Field Profiling

To enable field externalization, we collect two kinds of information: information that is gathered statically during the compilation of the instrumented program (e.g., information on objects in the image heap) as well as information that is gathered dynamically during the execution of the instrumented program.

The information that is computed at compile time is only available very late in the compilation process, e.g., after the compilation of all methods or when computing the layout of all types. Field externalization, however, has to be initiated early to adapt the corresponding types and field accesses. Therefore, we want to communicate such information from the first compilation (for the instrumented binary) to the subsequent, optimizing

Table 1 Overview of all metrics we collect. The *level* describes the granularity at which the tracking is performed.

Metric	Level	Description
allocations	Type	The number of allocations of a type.
size	Type	The exact and unaligned size of the instances of a type. Used to calculate how many bytes need to be externalized to achieve a reduction of the instance size for that type.
non-default value count	Field	The number of objects of a type that hold a non-default value for a specific field at any point during the execution of the program.
access kind	Field	Indicates whether the field is eligible, unused, or accessed via reflection, in an unsafe context, or in a VM-internal context.

compilation. We do this by storing compile-time profiling information in the compiled binary itself. When executing the instrumented binary, we collect the run-time profiling information and merge it with the static profiling information stored in the binary. This combined profiling information is then stored in the profiling file and subsequently used for optimizations in the final compilation of the program.

3.1 Information needed for field externalization

As it is expensive to write a non-default value to an externalized field, we need to make sure to only externalize fields where such writes happen rarely compared to the number of allocated objects of that type. Therefore, we need to know how many objects of a certain type are allocated. For each field, we also need to know in how many objects the respective field ever holds a non-default value. Furthermore, we need information about the size of a type, so that we can calculate how many bytes we need to remove from a type to achieve an actual size reduction (taking alignment into account). The points-to analysis (cf. Section 2.2.1) of GraalVM Native Image is able to detect fields that are unused; they will be removed automatically. We don't want to hinder that optimization and thus have to detect and prevent externalization of such fields. Finally, as we cannot safely externalize fields that are accessed in an unsafe context (via `sun.misc.Unsafe`) or in a VM-internal context, we also have to store information about incompatible accesses per field.

3.2 Metrics

We present our collected metrics in Table 1. The *level* indicates whether the metric is collected per type or per field. Details of the individual metrics are discussed below.

Allocations To accurately track the number of allocations per type, we consider both compile-time information and run-time information. As explained in Section 2.2.2, Native Image already allocates objects at compile time and stores them in the *image heap*. Thus, we count those instances before they are written into the final binary. We track the objects allocated at run time during garbage collection, i.e., for each object allocation, we increment the counter for the respective type.

Size The size of a type is computed late in the compilation process, whereas field externalization requires this information early in the compilation process. Hence, when compiling the instrumented program, we store the types sizes in the profiling data.

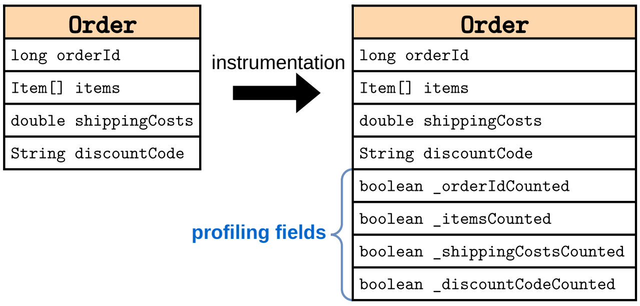


Figure 2 Instrumentation of the `Order` class from Figure 1 to track the *non-default value counts*. A profiling field is added for each original field.

Non-default value count We track this metric at field level. For each field, the metric represents the number of objects that—at any point during the execution of the program—held a non-default value in this specific field. For example, if a non-default value is written ten times to the same field for a single object, the *non-default value count* is only increased by 1. This metric is important for the selection of externalized fields, as only fields that have a low non-default value count—compared to the number of objects allocated of that type—are good candidates for field externalization. We do not count the total number of writes to a field, as a single non-default value write is enough to cancel out the achieved memory reduction for a single object due to the required companion object. Similar to the *allocations* metric, we also consider the objects in the image heap for this metric.

To identify first-time non-default value writes for each field/object combination, we instrument all types in the profiling run. For each field, we generate an additional boolean field that indicates whether a non-default value has already been written for that field/object combination, as can be seen in Figure 2. To correctly track the *non-default value count*, we have to instrument all field writes. The instrumentation of field accesses is depicted in Figure 3. First, we check whether the value written to the field is a non-default value, i.e., in the example, we check whether `x` is not `null`. In that case, we verify that the field has not yet been written for this object. If so, we set the profiling field to `true` and increment the counter associated with the *non-default value count* metric for that field.

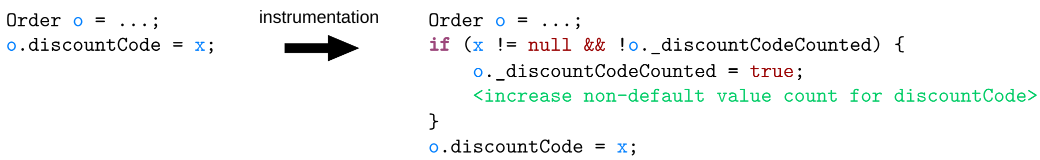


Figure 3 Instrumentation of a write to the `discountCode` field from the example in Figure 1.

Access kind As Native Image performs extensive static analysis, including points-to analysis (cf. Section 2.2.1), we are able to identify where and how fields are accessed. However, again, this information is only fully available late in the compilation process, and thus, similar to the size metric, we store that information in the profiling data when compiling the instrumented program. The access kind denotes one of three cases: *Unused*, when the field is unused and thus deleted by Native Image, *incompatible*, when the field is accessed using reflection, in an unsafe context or in a VM-internal context, or *eligible*, for fields that do not fall into any of the other categories. Only *eligible* fields are considered relevant for field externalization.

4 Field Externalization

The goal of field externalization is to reduce the memory footprint of an application by reducing the size of objects. To do so, we use heuristics based on the profiling information described before to identify fields where the default value dominates within types and to determine whether removing them from a type is beneficial. If we remove fields, we have to adapt their accesses within the compiled code: In case we ever write a non-default value, a companion object has to be allocated that consequently stores this field value. Subsequent accesses to externalized fields of the object then have to access this companion object. To actually benefit from field externalization, we need to minimize the number of required companion objects by tuning the heuristics accordingly.

4.1 Externalization Heuristic

Early in the compilation of a program, we load the profiling information generated during the instrumented run. For every type, we query the profiling data for its fields and use the metrics described in Section 3.2 to determine whether externalization is beneficial. Notably, we disallow externalization of types for which we do not have profiling data or for which the profiling reports no allocations. First, we determine a set F of all externalizable fields of a type as follows:

1. We only externalize fields where the *access kind* shows *eligible* accesses, i.e., we don't support externalization of *unused* or *incompatible* fields.
2. We determine the fraction of *non-default values* in all allocations, i.e., the percentage of instances where the field at some point held a non-default value. We compare this with a configurable threshold and mark the field for externalization if its non-default value fraction does not exceed this threshold:

$$\frac{\text{non-default value count}}{\text{allocations}} \leq \text{externalization threshold}$$

   Based on experiments, we use 5% as a default value for this threshold because it yielded the overall best results in our benchmark set.

While this approach yields all externalizable fields of a type, we still need to determine whether externalization of those fields actually shrinks the objects of this type. Consider the `Order` class from the initial example in Figure 1. Figure 4a depicts the initial memory layout of an object of this class: The object header typically takes up 8 bytes. *Compressed object pointers* [32, 62] reduce the size of reference fields to 4 bytes. As the figure shows, this results in an overall size of 32 bytes. Native Image aligns objects at 8-byte boundaries [62]; with a size of 32 bytes this requirement is already met. The requirement for externalization is to reduce the size to at least the next smaller alignment boundary (24 bytes). Otherwise, the reduction in the type size is canceled out by the alignment. In the example, this means that we have to reduce the overall size by 8 bytes. We have to take into account that our externalization approach injects a new reference field (4 bytes) into the type that points to a companion object if this is needed. Therefore, the required reduction in the example increases to 12 bytes.

Figure 4b shows the result of externalizing the `shippingCosts`, an 8-byte `double` field. Unfortunately, if we consider the 4 bytes of the injected `_ref` field, the object size is only

object structure	size (bytes)
object header	8
long orderId	8
Item[] items	4
double shippingCosts	8
String discountCode	4

before alignment	32
after alignment	32
req. externalization	12

(a) Layout of an `Order` object from Figure 1 assuming an 8 byte object header, *compressed object pointers* [32, 62], and 8 byte alignment [62].

object structure	size (bytes)
object header	8
long orderId	8
Item[] items	4
double shippingCosts	0
String discountCode	4
Order$Companion _ref	4

before alignment	28
after alignment	32
size reduction	0

(b) Layout of `Order` where `discountCode` has been externalized but overall object size remains the same due to the required addition of a `_ref` field.

object structure	size (bytes)
object header	8
long orderId	8
Item[] items	4
double shippingCosts	0
String discountCode	0
Order$Companion _ref	4

before alignment	24
after alignment	24
size reduction	8

(c) Layout of `Order` where `discountCode` and `shippingCosts` have been externalized, resulting in a reduction of the object size by 8 bytes

Figure 4 Memory layout of the `Order` class from Figure 1 before and after externalization.

reduced to 28 bytes and alignment brings this up again to 32 bytes. The situation is different when externalizing both `shippingCosts` and `discountCode`, as shown in Figure 4c: The overall reduction (including the injected field) is now 8 bytes, which reduces the object size to 24 bytes—again at an 8-byte-alignment boundary.

Profiling information gives us the size t_size of each type before alignment (cf. Section 3.2). Assuming the injected field size to be ref_size, and an alignment of 8 bytes, the minimal required amount of bytes that must be externalized is calculated as follows:

$$\text{min. externalization bytes} = ref\_size + \begin{cases} 8, & \text{if } t\_size \bmod 8 = 0 \\ t\_size \bmod 8, & \text{otherwise} \end{cases}$$

Then, we use the following formula to determine whether externalization is beneficial for the set of theoretically externalizable fields F that we derived in the prior step:

$$\sum_{f \in F} sizeof(f) \geq \text{min. externalization bytes}$$

If the result is to externalize the fields F, we subsequently delete them from the type. For each type with externalized fields, we create a corresponding *companion type*, a synthetic class that only contains the externalized fields of the class. Subsequently, we inject the `_ref` field—also called *companion reference field*—that may reference a companion object if needed into the type with externalized fields.

4.2 Rewiring Accesses to Externalized Fields

Externalizing fields into companion classes is only the first step in the externalization process. Next, we have to adapt all accesses to externalized fields, as shown in Figure 5. Values of externalized fields are now stored in the corresponding fields of the companion object. This object, however, should only be allocated if we write a non-default (non-zero/non-null) value to an externalized field. Therefore, regardless of the access kind, we have to make a case distinction on whether a companion object exists. For each object, we store this information in an additional header bit, which we call the *companion allocated (CA)* bit. If the bit is

```
Order o = ...;
var x = o.discountCode;
```

(a) Read access to `discountCode` before field externalization.

```
Order o = ...;
o.discountCode = y;
```

(b) Write access to `discountCode` before field externalization.

```
Order o = ...;
var x;
// check whether companion exists
if (<ca_bit_set>(o))
  x = o._ref.discountCode;
else
  x = null; // use default value
```

(c) Read access to `discountCode` after field externalization. Since the field no longer exists within the original type, we either yield the default value or have to get the field value from the companion object depending on the result of the CA bit check.

```
Order o = ...;
if (<ca_bit_set>(o)) { // check whether companion exists
  o._ref.discountCode = y;
} else if (y != null) { // default value check
  var _cmp = new Order$Companion(); // allocate companion
  _cmp.discountCode = y; // write companion field
  o._ref = _cmp; // store companion in original object
  <set_ca_bit>(o); // mark existence of a companion
} // no write if we write the default value
```

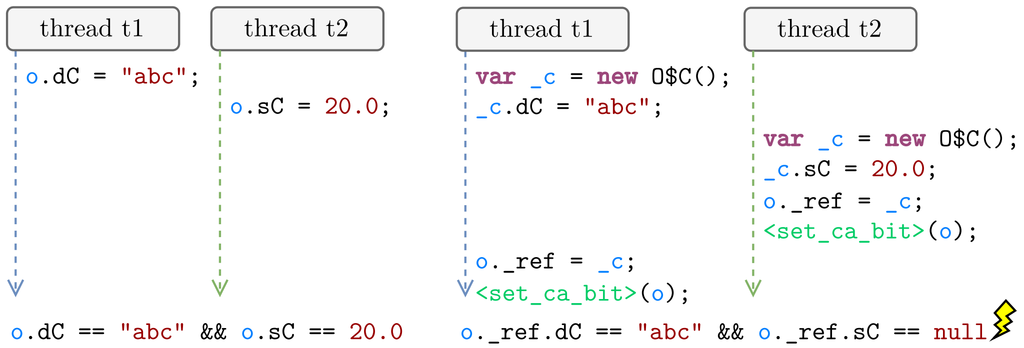
(d) Write access to `discountCode` after field externalization. Depending on whether the CA bit is set, we may have to allocate a companion object if the value to write is not the default value.

Figure 5 Accesses to fields of the `Order` class before and after field externalization.

unset, no companion object exists and the fields never held anything other than the default value. This is the ideal case and our heuristics mentioned in Section 4.1 are tuned to ensure that this is the most frequent case. Hence, we have to adapt the field accesses in such a way that this case is still efficient. If the bit is set, it indicates that an object already has a companion object. This should be the outlier case which we want to avoid as much as possible, as the companion object consumes additional memory.

Read Accesses Figure 5a shows a read access to the `discountCode` field of the `Order` class. Figure 5c depicts the adaption of the read access when the field is externalized. At every access to an externalized field, we have to introduce a header bit check (CA bit) that tells us whether the object already has a companion object. If the bit is unset, the read is trivial as this tells us that the field still holds its default value. This is the fast path and should be the default case. Thus, we can simply yield the default value as a result of the read access. If the bit is set, the companion object must exist and we have to load it from the `_ref` field to get the actual field value.

Write Accesses For write accesses, as shown in Figure 5b, we need to do more work, as each write may require the allocation of a companion object. Figure 5d depicts the adaption of write accesses to an externalized field of the `Order` class. Again, the CA bit tells us whether the object already references a companion object. If a companion object exists, we write the new value to the corresponding companion object field. If the bit has not been set, we have to check the value that is written. If we again want to write the default value (null or zero) to this field and the companion object does not exist yet, we can simply skip the write as the field is guaranteed to still hold the default value. This is the optimal case and should be the most frequent path. However, if we write a non-default value, we have to allocate a new companion object and write the value to the field in the companion object. Additionally, we have to store the companion object in the original object and set the CA bit to communicate this change to future accesses. This is the slow path. After the companion object has been allocated, all the benefits that we achieved for this particular object by reducing its size are voided. Thus, our heuristics should minimize the frequency of this case.



(a) Write access to `Order` fields from multiple threads before field externalization.

(b) Write accesses to `Order` fields from multiple threads after field externalization if the companion object was not created before. As two threads may simultaneously allocate companion objects and store them in the original object, the result is a lost update as the companion object from thread `t2` is overwritten by the one from thread `t1`.

Figure 6 Write access in multiple threads to fields of the `Order` class before and after field externalization. For brevity, we abbreviate the fields `discountCode` and `shippingCosts` with `dC` and `sC`, and the companion class `Order$Companion` with `O$C`.

4.2.1 Adhering to the Java Memory Model

The adaptations necessary to handle accesses to externalized fields are trivial in principle. However, it gets more complicated when we consider their effects in the context of the Java Memory Model, which defines the language semantics in multithreaded environments [30]. Particularly, the writing process is affected, as a write to an externalized field may now consist of multiple instructions, which may interfere with each other if executed on the same object in different threads. Figure 6a shows the behavior of two write accesses to fields of the same object in multiple threads without externalization. After the writes have been completed, both values are eventually [1] visible in the object. Figure 6b shows what could happen when writing to externalized fields: Here, we assume that the target object does not have a companion object yet—the CA bit is unset—and both threads want to write non-default values to two independent fields, forcing allocation of a companion object. *Thread t1* first allocates a companion object and stores a value to the first field. Then, *thread t2* resumes, also allocates a companion object and stores a value to the second field. In the example, *t2* immediately stores the companion object in the original object and sets the CA bit. When *t1* resumes, it also stores its companion object in the original object and thus overwrites the `_ref` field. This results in a *lost update* as the value written in thread `t2` is lost. In fact, the order of the writes does not even matter: the allocation of two different companion objects for the same original object is a problem in terms of the memory model. Therefore, we need to introduce a synchronization mechanism for externalized field writes.

As read accesses can simply rely on the CA bit to determine whether a companion object exists, they require no modification. If another thread writes a non-default value to the same field at the same time and thus allocates a companion object, this would result in a race

[1] Omitting the fact that non-volatile writes within threads may not be visible to other threads immediately [30]—the overall problem still remains.

condition in the case of a non-synchronized access even without field externalization. Thus, we can treat this simply as a race condition.

Figure 7 describes the synchronization process we introduce to correctly mimic field write semantics in our `Order` example. The dashed green path denotes the fast path and thus the most frequent one. The initial steps remain the same: We check whether a companion object exists (via the CA bit). If so, we write the value into the companion object. Otherwise, we check whether the write concerns the default value. If that is the case, the write can be omitted. The process differs when we write a non-default value and thus require a companion object. First, we allocate the companion object (**A**) and assign the field value (**B**). Note that this can be performed by multiple threads at the same time per object. Next, however, we use an additional header bit—the *companion-in-creation (CC)* bit—that indicates whether some other thread currently allocates the companion. We use a *compare-and-set (CAS)* instruction [20] (**C**) that atomically tries to set the CC bit if and only if it has not been set. A CAS is a three-operand instruction that takes an *expected value* (the object header where the CC bit is zero), a *destination operand* (the address of the object header) and a *source operand* (the object header with the CC bit set). Then, it performs the following steps atomically: First, it compares the *expected value* with the value in the *destination operand*. If the values match (the object header does not have the CC bit set), it writes the value in the *source operand* to the *destination operand* (sets the CC bit) and returns the *expected value*. If the values are different, it returns the value in the *destination operand* (the actual object header). Subsequently, we can use the return value of the instruction to determine whether the current thread could set the CC bit. Therefore, the CAS can only ever succeed for a single thread per object. We call this thread the *CC thread*. The CC thread subsequently enters a *protected region*, where it can safely assign the companion object it previously created (**D**) and also set the CA bit (**E**) to signal that a companion object is available now. Any other thread that tried to CAS the CC bit at the same time has to wait until the CC thread has completed the assignment of the companion. We introduce a *spin lock* [21, 50] (**F**) that loops until the CA bit is set. Note that this branch is entered very infrequently, namely only if two threads try to allocate a companion object for the same object at the same time. Once the thread leaves the spin lock, it can safely assume that a companion object exists and can thus access it and perform its field write (**G**).

4.3 Compile-Time Externalization

As explained in Section 2.2.2, objects that can be created at compile time, but are needed at run time, are already allocated during compilation and are stored in the image heap. Our externalization process, however, only affects the objects that are created at run time—objects allocated at compile time still adhere to their original layout. To ensure that the objects in the image heap are compatible with the types after externalization, we introduce special handling for all compile-time-allocated objects that have externalized fields. When writing those objects into the image heap, we check whether they contain a non-default value for an externalized field. If so, we allocate a companion object and copy the values of the externalized fields to the companion object. Finally, we link the companion object with the original object (now reduced to its non-externalized fields and the companion reference) and also store the companion object in the image heap. Thus, the object representation in the image heap is compatible with the types after externalization.

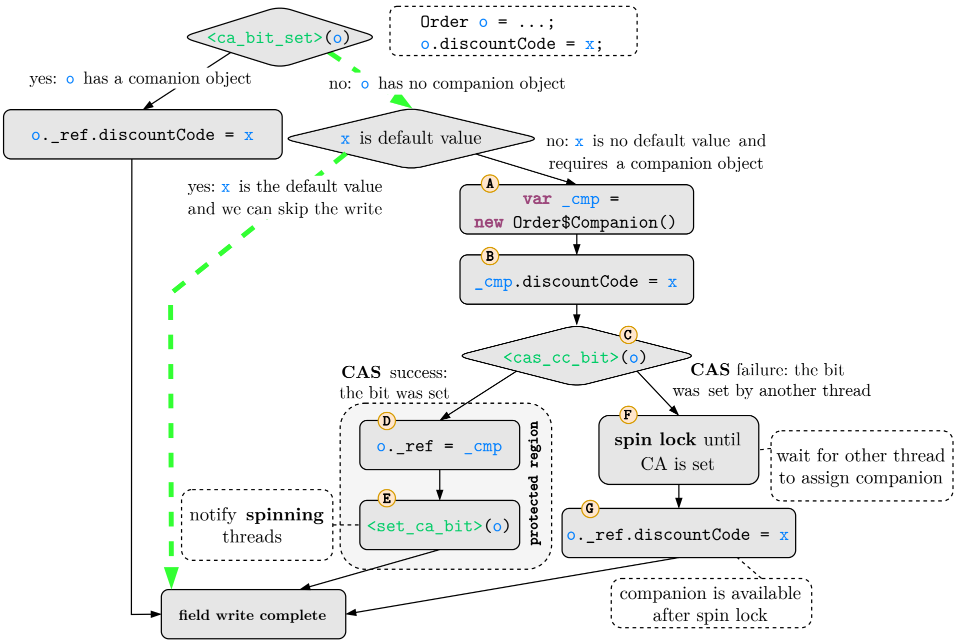


Figure 7 Write accesses to externalized fields including a synchronization mechanism that uses 2 header bits (*CA*, *CC*) to ensure that for a specific object only one thread can assign the companion object. The dashed (green) path is the (most frequent) fast path.

4.4 Field Externalization and Class Inheritance

Consider the case when a class `A` has some fields that can be externalized. Suppose that a subclass `B` of `A` also has externalizable fields, making it eligible for externalization according to our heuristic. With our current approach, B would require two companion reference fields (for `A$Companion` and for `B$Companion`), and thus 4 header bits (two bits per reference field). Hence, field externalization does not scale efficiently with class hierarchies. Furthermore, the number of object header bits that can be used for field externalization is limited, hence, preventing us from using more than 2 object header bits. However, as externalizing fields of classes that already have externalized superclasses is beneficial (as it can further shrink the object size), we want to support it as well.

4.4.1 Companion Inheritance

We present the solution to externalizing fields of class hierarchies based on the example in Figure 8. The example shows three classes `A`, `B`, and `C`, where `A` is the superclass of `B` and `B` is the superclass of `C`. In the example, two fields of class `C` can be externalized. However, the superclass `A` also has externalized fields. To solve this problem in an efficient manner, we propose that the companion type that is created for `C` (`C$Companion`) is defined as a subclass of the companion type of the (closest) externalized superclass (here `A$Companion`). Thereby, the problems mentioned above are solved, as at most one companion object is needed for an object. The `C$Companion` instance now contains both: the fields externalized in `C`, as well as the fields externalized in `A`. Therefore, only a single companion reference field and 2 object header bits are needed.

However, this approach introduces an additional challenge: We can no longer determine at compile time which companion object should be allocated when writing to an externalized field. For example, if a method has a parameter of type `A` and subsequently writes a non-default value to the `a1` field, we would either have to allocate an `A$Companion` object or a `C$Companion` object depending on the actual object. Thus, the allocated companion object now depends on the *dynamic type* of an object. We solve this by introducing a run-time type check on the object, but this introduces overhead in terms of performance and code size.

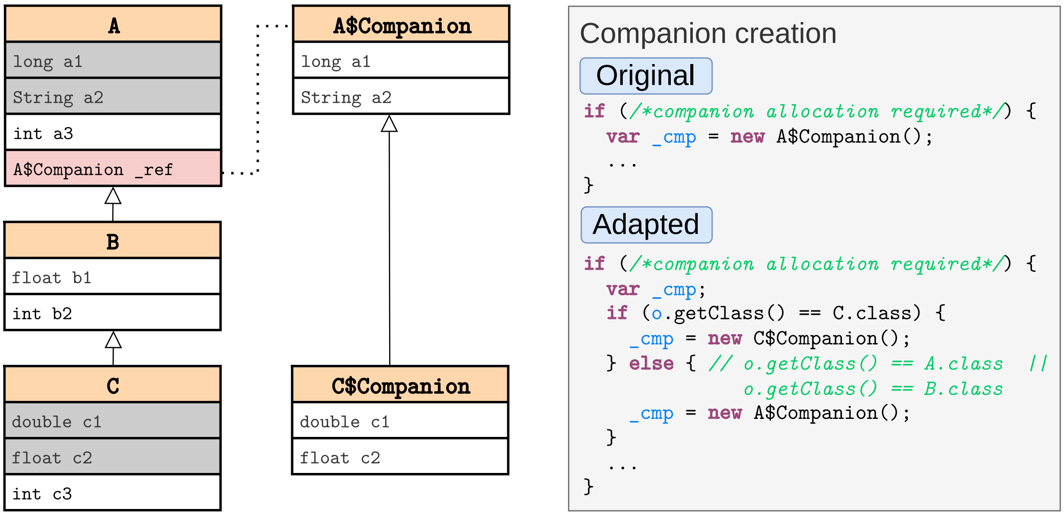


Figure 8 Companion hierarchy for a class hierarchy with multiple types with externalized fields. The necessary adaptations to the companion creation process are illustrated on the right-hand side.

4.4.2 Companion Factory Methods

The companion creation code gets more complicated the more classes of a class hierarchy are externalized. As we need to adapt every write to an externalized field, directly inserting all type checks and allocations of the right companion object within the class hierarchy leads to code bloat. Therefore, we utilize what we call *companion factory methods*. A *companion factory method* is a generated method that receives a type with externalized fields as a parameter and returns a new instance of the matching companion type. Hence, for each root of an externalization hierarchy one companion factory method is generated. That method contains the if-else-if ladder for all externalized subtypes of the externalized root type and returns a companion object of the respective type.

Figure 9 shows an example of a companion factory method and its usage. The *companion factory method* `createCompanionA` is used to create the correct companion object based on the class passed as a parameter. During compilation, we insert a call to this method when writing a non-default value to an externalized field, as shown on the left-hand side of Figure 9. Through these factory methods we can reduce the code bloat. We prevent inlining of these methods to not mitigate their effect but also inform the compiler of their effects to not prevent other optimizations such as partial escape analysis [52].

4.5 Masked Companion References

Our field externalization approach requires adding a reference to a companion object to enable us to handle the cases when non-default values are written to externalized fields. Therefore, externalization also needs to amortize the cost of that additional field, as explained

```
...
if (/*companion allocation required*/) {
  var _cmp = createCompanionA(o.getClass());
  ...
}
...
```

```
static A$Companion createCompanionA(Class c) {
  if (c == C.class){
    return new C$Companion();
  } else { // c == A.class || c == B.class
    return new A$Companion();
  }
}
```

Figure 9 Creation of a companion object using a *companion factory method*.

in Section 4.1. Consequently, this limits the applicability of our approach and thus our optimization potential.

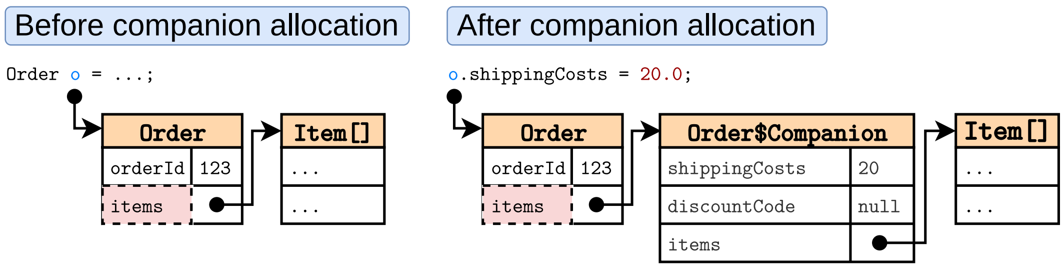


Figure 10 As long as no companion object is need for an object with externalized fields, the *masked companion reference* field (`items`) holds its normal value. When a companion object is needed, the value of the `items` field is evacuated into the companion object and the reference to the companion object is stored in the `items` field.

To tackle that issue we propose an optimization of field externalization, which we call *masked companion references*. A *masked companion reference* is a non-externalized *reference* field of a type that we reuse as the companion reference field. As long as no companion object is needed, this field is used just like a normal field and stores its normal value. In Figure 10, we again show the `Order` example from the previous sections with the two externalized fields `shippingCosts` and `discountCode`. The remaining `items` field is used as the *masked companion reference*, indicated by the dashed line and light red background. As shown in the first part of the image, the `items` field holds a reference to an ordinary item array, because no companion object is required yet. When a non-default value is written to an externalized field, a companion object is allocated (as explained in Section 4.2) and the reference to the companion object is stored in the `items` field. However, before writing the companion reference, the old value stored in the `items` field needs to be evacuated in order to preserve its value. Therefore, when using *masked companion references*, the companion object contains not only the externalized fields but also the *masked companion reference* field. In the example of Figure 10, when a non-default value is written to the externalized field `shippingCosts` (shown at the right-hand side of the figure), a companion object is allocated and the value that should be stored in `shippingCosts` is stored there. Then, the value stored in `o.items` is evacuated and stored in the companion, and finally, the reference to the companion object is stored in the `items` field (the masked companion reference field) of the `Order` object. By applying this optimization, we do not need to introduce an additional companion reference field that increases the size of all `Order` objects just to hold the reference to an—ideally—rarely needed companion object. If a type does not contain a suitable reference field, we can still fall back to the original approach.

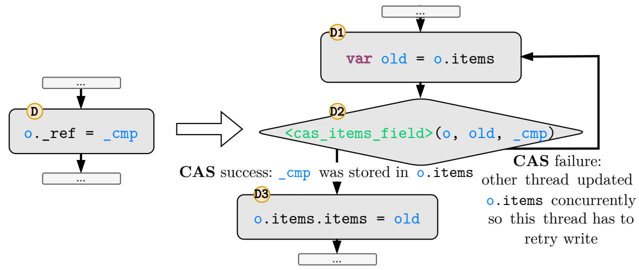


Figure 11 Changes necessary to the process shown in Figure 7 at step **D** for externalized field writes with masked companion references.

4.5.1 Field Access Adaptations for Masked Companion Reference Fields

To accommodate masked companion reference fields we also have to slightly change the field access pattern introduced in Section 4.2.1. First, both field writes and reads now have to access the corresponding masked companion reference to load a companion object if required—the check for its existence is still done via the CA bit. Figure 11 shows the change necessary for writing to an externalized field, as explained in Figure 7. We now no longer have a dedicated companion reference field that points to a companion object. Thus we have to consider both cases: another thread writing a new `Item[]` instance to `Order.items`, and another thread writing a non-default value to an externalized field and thus assigning a newly allocated companion object to the masked companion reference field. Therefore, we have to ensure that both cases are properly synchronized. As step **D** of Figure 7 is in the protected region, we can at least be sure that no other thread concurrently tries to write a companion object. However, regular writes to the masked companion reference field are not guarded by the CC bit. Therefore, we have to make sure to check for concurrent writes when storing the newly allocated companion object. We first load the original value from the companion reference field `items` (**D1**). Then, we use a CAS instruction (**D2**) to atomically check, whether the `old` value in `o.items` has changed. If not, the CAS instruction stores the companion object (`_cmp`) in the `items` field. The result of the CAS tells us, whether the companion object was stored. If the value did change between reading the field and the CAS (and the companion object was not stored), we repeat steps **D1** and **D2** until they succeed. While this is an expensive process, we expect it to be very infrequent. After the CAS, we still have to store the old value of the masked companion reference (`old`) in the corresponding companion object field (`o.items.items` in **D3**).

Writing to a Masked Companion Reference Field

Since the masked companion reference field is a regular field within the original object, we have to adapt accesses to it. When writing to the field, we have to check whether it already references a companion object. The necessary steps are outlined in Figure 12. Once again, the—ideally most frequent—fast path is highlighted with dashed (green) lines. In step **A**, we load the value of the masked companion reference field `o.items`. The CA and the CC bits tell us whether the loaded value might already be a companion object (**B**). If one of the bits is set, we specifically check whether the CA bit is set (**C**). If not, we know that the CC bit is set and that another thread is currently assigning the companion object to this object. Hence, we *spin lock* until this process is complete (**D**). If the CA bit was set or after the spin lock, we know that a companion object must exist and thus assign the value to the

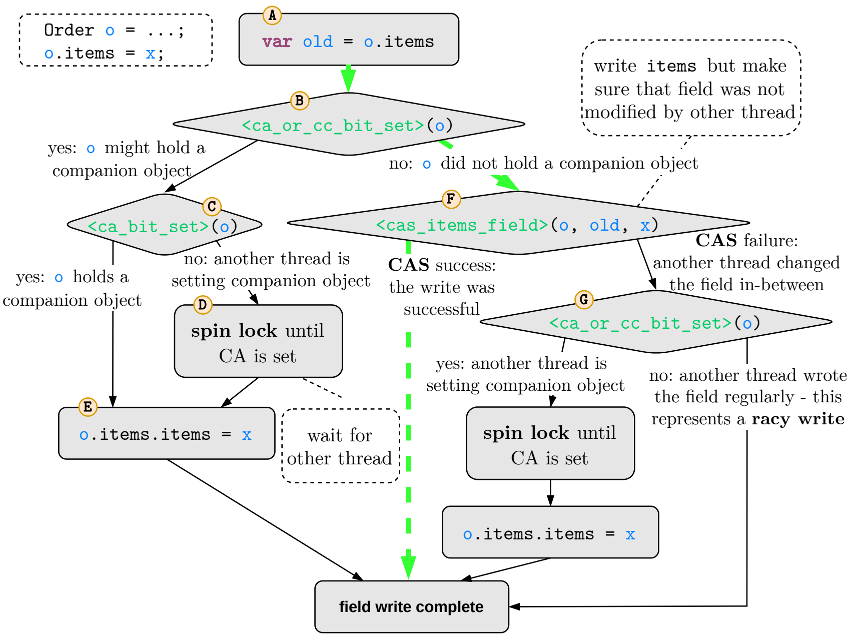


Figure 12 Write access adaptations to masked companion reference fields that take into account multithreaded accesses. As a masked companion reference field can be used both for regular writes and for storing the companion object, synchronization is more expensive than for regular externalized field writes.

corresponding `items` field *within* the companion object (**E**). Note that we cannot use the value read before from the masked companion reference field (`old`), as this value may not have been a companion object at that point. Therefore, we re-read the companion object and assign the actual value (`x`) to its field (**E**).

If step **B** showed that no bit was set, we try to directly store the value in the masked companion reference field (**F**). As this may again interfere with similar accesses in other threads, we use an atomic CAS instruction. This CAS compares the previous value in the masked companion reference field (`old`) with the value currently stored in the field. If this comparison succeeds, it stores the desired value `x` in the masked companion reference field. In this case, we are done with the field write. If the CAS did not succeed, we have to perform a similar procedure as in **B**, as some other thread has changed the value of the field. We cannot know whether the field still holds a *regular* value or a freshly allocated companion object and thus have to check the header bits again (**G**). If any bit was set, we make sure to wait until the CA bit is set (which ensures that the other thread successfully wrote the companion) and only then write the value to the companion object. If no bit was set, we know that the other thread writing the same field wrote another regular value (i.e., some other `Item[]` array) to the field. We consider this a race condition (*racy write*) and can thus skip updating the field.

Reading from a Masked Companion Reference Field

In contrast to read accesses to externalized fields, reading from a masked companion reference field also requires some synchronization, as another thread may concurrently assign a companion object to the same field. We discuss the necessary steps in this process based on Figure 13—the dashed green path is the fast path. First, we load the current value of

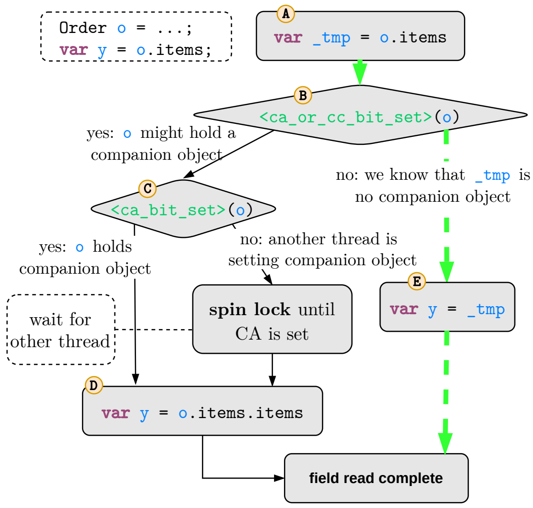


Figure 13 Read access adaptations to masked companion reference fields that take into account multithreaded accesses.

the masked companion reference field (`o.items`). Then, similar to the writing process, we check for either the CA or the CC bit, as either tells us that the previously loaded value might be a companion object. If any bit is set, we specifically check the CA bit (**C**) and then either first wait for it to be set or immediately access the corresponding field within the—now safely available—companion object (`o.items.items` in **D**). If no bit was set, we know that the loaded value from step **A** cannot be a companion object and thus, we simply use this value as a result of the field read (**E**).

5 Evaluation

We conducted our evaluation on a comprehensive, mixed benchmark corpus. This corpus blends classical benchmark suites (*DaCapo 9.12-MR1-bach* [5], *Scala-DaCapo 0.1.0* [48], *Renaissance 0.9.0* [40]) with additional benchmark suites designed for Native Image performance assessment, all built atop leading Java microservice frameworks—namely Spring [59], Quarkus [42], and Micronaut [44]. Note that we excluded standard benchmarks that are not supported by the version of Native Image on which our approach is based. Furthermore, our experiments revealed an infrequent bug within GraalVM Native Image related to inlining that distorted our measurements for the *scala-dacapo factorie* benchmark. This bug was reported but has not been fixed yet. Thus, we exclude the benchmark from both analysis and evaluation. Each framework-based benchmark suite includes a concise “helloworld” scenario based on its respective launcher or “getting started” material, thus showcasing the fundamental functionalities of the framework. In addition, every suite contains a second, more complex benchmark with varying workload parameters (tiny, small, medium, etc.). Specifically, the Spring suite employs a tailored version of the Spring Boot PetClinic Sample Application [60]. Quarkus incorporates a microservice benchmark originating from the Apache Tika Quickstart [41]. Micronaut, in turn, provides a second benchmark known as *ShopCart*, a web shopping application.

Benchmark Methodology

We compared the standard version of GraalVM Native Image to our modified version of Native Image, which automatically externalizes rarely used fields. Both configurations involved an initial profiling run with a slightly reduced workload for PGO, followed by the actual measurement run for each benchmark. We executed each configuration 8 times—each execution involved a new profiling run to get new profiling data. The experimental platform was an Intel I7-4790K @ 4.4 GHz with 20 G of main memory. Hyper-threading, frequency scaling, and network access were disabled. All benchmarks were performed using Java 21.

5.1 Benchmark Results

We present the results of the evaluation of our approach on standard benchmarks in Figure 14. Figure 15 contains the results on microservice benchmarks. First, we compare the number of overall allocated bytes compared to the baseline (*relative allocated bytes*). It is important to note that the microservice benchmarks are *throughput* benchmarks, i.e., they are executed for a fixed duration and we measure how many *requests* they manage to process (as they all concern web frameworks). The number of requests furthermore represents the performance metric for these benchmarks (compared to the benchmark *time* metric in standard benchmarks). Hence, the allocated bytes in a benchmark execution also depend on the number of processed requests. Thus, we normalize the *relative allocated bytes* by the number of processed requests. This is represented by the *allocated bytes/request* metric (row 3 in Figure 15). In addition to the allocated bytes, we also compare the execution times/throughput as well as the maximum resident set size (RSS) of each benchmark.

Allocated Bytes

Because our focus lies on reducing allocated memory, we monitor the total allocated bytes for each benchmark by summing the sizes of newly allocated objects. This metric is also used in related work to evaluate the effectiveness of field externalization [3]. The total allocated bytes are our most stable metric (Figure 14). For this metric, almost all standard benchmarks show reductions, namely 2.72% on average for *dacapo*, 1.30% for *scala-dacapo*, and 3.60% for *renaissance*. The *renaissance* benchmarks *philosophers* and *scrabble* benefit the most from our approach. In both benchmarks, objects with externalized fields appear frequently and contribute significantly to the overall allocated bytes. At the same time, they do not show many companion objects.

While the overall allocated bytes are also improved in all microservice benchmarks, the normalized metric (*allocated bytes per request*) shows a minor regression in the *tiny* workload of *quarkus tika* (Figure 15). This regression stems from a performance degradation (around 6.52%) that outweighs the improvement in allocated bytes (5.64%).

The noticeable differences in the results in the *spring petclinic* benchmarks (Figure 15), where we see large improvements for *huge* and *large* (35.15% and 20.53%, respectively), but only minor improvements for the three smaller workloads, can be explained by the single type `java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject` that shows large fluctuations in its allocations. In the baseline, millions of objects of that type are allocated in each run for all workload sizes. With our approach, we still see millions of allocations for this type in the smaller workloads but only around 22000 allocations in *huge*, thus explaining the significant improvement here. In the *large* workload, the allocation count fluctuates between the baseline values and the 22000 objects, hence the large error.

RSS

Although we use the resident set size to estimate memory consumption, we observed high RSS fluctuations in many benchmarks, as shown in Figure 14 and Figure 15. This is because this metric is highly dependent on the garbage collection behavior in a benchmark. Thus, minor variations in the run-time behavior can lead to different RSS values even for the same application. Our evaluation shows an average max-RSS reduction of 2.55% on standard benchmarks and a reduction of 2.45% on microservice benchmarks.

Most notable are the results in the *renaissance fj-kmeans* and *scala-doku* benchmarks (21.97% and 15.13% reductions, respectively)—despite a considerable error, their results are consistently below the baseline. Other *renaissance* benchmarks also show high errors: *mnemonics* seems marginally improved, while *par-mnemonics* shows a considerable regression.

For benchmarks from the *dacapo* and *scala-dacapo* suites, we see mixed results—most benchmarks (e.g., *fop*, *pmd*, *scaladoc*, *tmt*) show minor reductions of the max-RSS, while others (e.g., *lusearch*, *sunflow*, *apparat*) exhibit minor regressions.

In the microservice benchmarks, the results for the *spring petclinic* benchmarks are most promising, with an average reduction of 6.08% and only a minor regression in the *helloworld* benchmark. While our approach also reduced the max-RSS in the *small* workload of the *quarkus tika* significantly, the other benchmarks of the suite show minor regressions. Similarly, the *quarkus* benchmarks mostly show regressions (0.90% on average).

Auxiliary Metrics

As mentioned in prior sections, we had to introduce additional checks for accesses to externalized fields and masked companion reference fields to ensure validity. Therefore, we expect a performance impact. Indeed, the standard benchmarks show a slight regression in terms of performance (2.14% on average, Figure 14). For most benchmarks, this performance impact correlates with their improvement in terms of allocated bytes. The most notable exceptions are *scala-dacapo scaladoc*, *renaissance finagle-chirper*, and *renaissance finagle-http*. In all three benchmarks, we actually see many externalized objects. Despite that we also see that our approach increases the allocation counts for some of the most used types or introduces allocations of types that we do not see in the baseline runs. These factors may points towards compilation issues: As we insert additional code at field accesses, we may exceed budgets for certain compiler optimizations such as inlining or loop optimizations. Similarly, we might also prevent escape analysis and scalar replacement of certain allocations. As many of these optimizations also are based on heuristics, these individual regressions might by solved by retuning these heuristics for our approach.

Interestingly, the *renaissance scala-doku* benchmark shows an improvement in terms of run-time performance combined with improvements in other metrics as well. The performance of the microservice benchmarks is also impacted; the number of processed requests is decreased by 6.09% on average (Figure 15).

In addition to the charts presented above, we also compare the size of the generated image (the binary size). As we introduce additional code (cf. field access modifications in Section 4.2), new types (companion types), and new methods (cf. companion factory methods in Section 4.4.2) into the compiled binary, we expect a slight increase in the image size. Our evaluation shows that the image size increases by 2.16% on average for standard benchmarks and by 3.36% on average for microservice benchmarks.

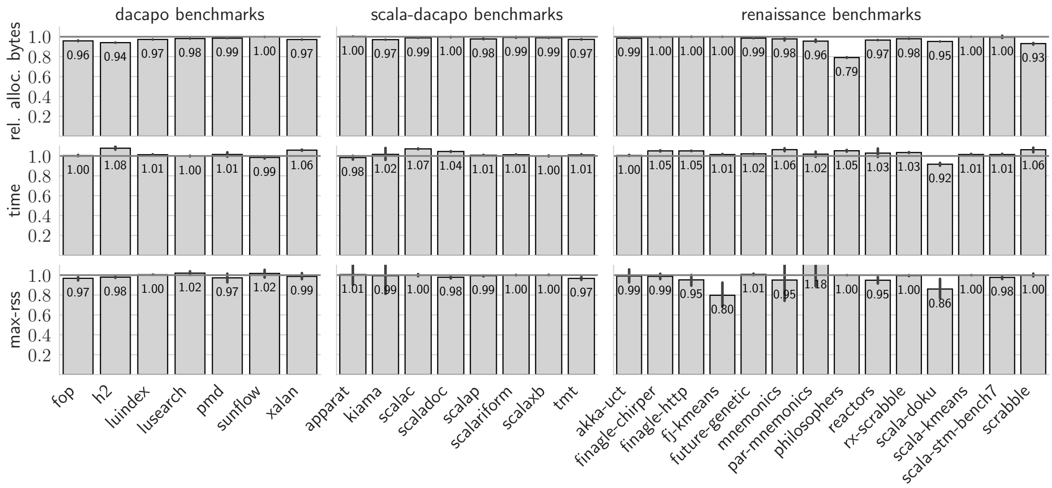


Figure 14 Evaluation of total memory allocation (*relative allocated bytes*), benchmark run time (*time*), and the max-RSS (*max-rss*) on standard benchmarks relative to results on Native Image without our approach.

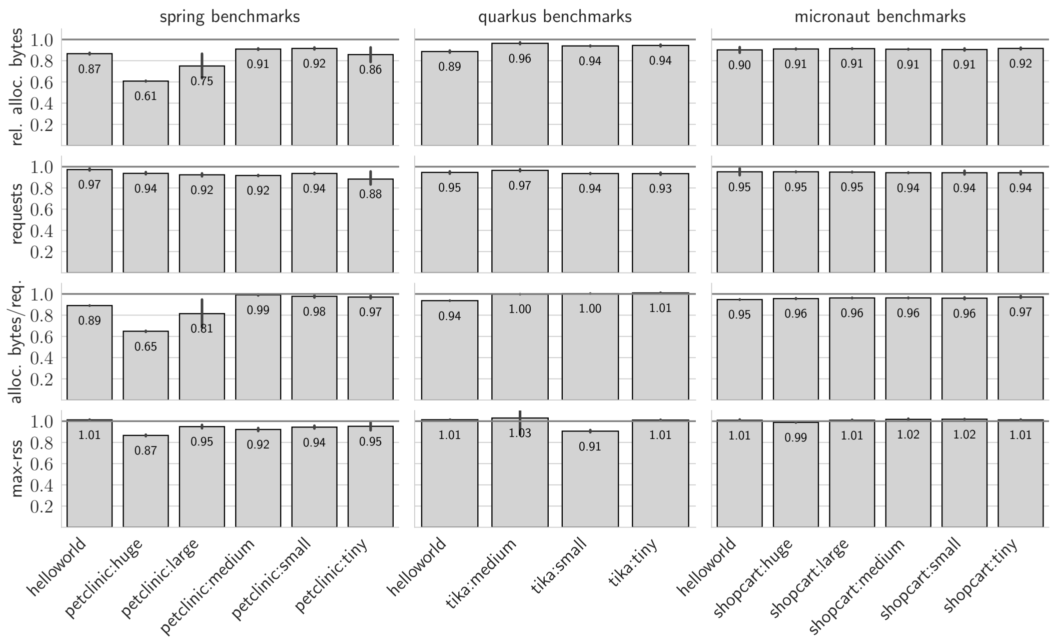


Figure 15 Evaluation of total memory allocation (*relative allocated bytes*), benchmark throughput (*requests*), memory allocation normalized by the performed requests (*allocated bytes/request*), and the max-RSS (*max-rss*) on microservice benchmarks relative to results on Native Image without our approach.

Externalization Analysis

To evaluate our externalization approach, including our heuristic, we also evaluated how many objects are affected by field externalization and how many companion objects are created. Table 2 presents our findings for each benchmark. The *externalization ratio* specifies the ratio of objects in the benchmark that had externalized fields, whereas the *companion object ratio* represents the ratio of objects with externalized fields that required a companion object. The ratio of externalized objects varies greatly across benchmarks. For example, there are nearly no externalized objects in *scala-dacapo apparat*, whereas in *renaissance scrabble* 33.4% of the allocated objects have externalized fields. Overall, there is a tendency that benchmarks with a higher externalization ratio profit more in terms of reduction of total allocated bytes from field externalization, as shown by our memory measurements presented above. However, there are outliers as well: For example, the *dacapo xalan* benchmark is the benchmark with the second highest *externalization ratio*, but the allocated bytes are only reduced by 3% and the max-RSS is only reduced by 1%. The reason for the low impact in this benchmark is that the objects with externalized fields only make up a minor part of the overall memory consumption. Thus, big parts of the allocated memory are unaffected by field externalization. Furthermore, the per-object memory savings are quite low in this benchmark, i.e., only few fields are externalized for the types with the most allocations.

The *companion object ratio* should be quite low—in general below 5%, as that is the threshold of our heuristic(cf. Section 4.1). However, as will be discussed in Section 6 there could still be cases with higher companion object ratio even if our profiling information is accurate. In our results, *renaissance rx-scrabble* has the highest *companion object ratio* with 6.5%. We found that `java.util.stream.SliceOps$1`, which is a subclass of the externalized class `java.util.stream.AbstractPipeline` is responsible for creating more than 99% of the companion objects allocated in this benchmark. More specifically, all `java.util.stream.SliceOps$1` instances wrote non-default values to externalized fields and thus required the creation of companion objects. In theory, our heuristic should prevent such cases by not performing externalization when the *non-default value count* is too high. However, our heuristic can only consider the data gathered during profiling, and in this case the ratio of `java.util.stream.SliceOps$1` objects was lower compared to the other subclasses of `java.util.stream.AbstractPipeline` (which did not trigger as frequent companion object allocations) in the profiling run than in the actual benchmark run. One potential reason for this inaccurate profiling information is the reduced workload in the profiling run. We made a similar observation in the *petclinic* benchmarks. Here, the culprit is the `org.h2.mvstore.Page` class. We externalize almost all fields of this abstract class because the profiling data reports low usages. However, in the benchmark, we actually allocate a companion object for every `org.h2.mvstore.Page` object, suggesting that the objects of this class behave very differently in the real benchmark run compared to the profiling run. Despite these issues, the benchmark still shows good results across all workloads.

6 Limitations

The main limitations of our approach concern the run-time behavior of companion objects and the expressiveness of the profiled heuristics in light of some corner cases. Also, we disallow externalization of specific field kinds by default.

Table 2 Evaluation of the ratio of objects with externalized fields (externalization ratio, **ER**) compared to the number of total allocated objects. For the companion object ratio (**COR**), the amount of externalized objects that used a companion object was calculated.

Suite	Benchmark	ER	COR
dacapo	fop	9.4%	0.5%
	h2	26.0%	1.4%
	luindex	25.3%	0.0%
	lusearch	9.0%	3.9%
	pmd	4.3%	4.2%
	sunflow	0.3%	0.1%
	xalan	33.3%	1.3%
scala-dacapo	apparat	0.0%	2.1%
	kiama	9.6%	2.9%
	scalac	3.7%	3.4%
	scaladoc	4.3%	2.2%
	scalap	1.0%	1.1%
	scalariform	1.5%	0.8%
	scalaxb	2.7%	0.6%
	tmt	13.8%	0.0%
renaissance	akka-uct	1.9%	0.0%
	finagle-chirper	5.9%	0.1%
	finagle-http	7.4%	0.0%
	fj-kmeans	2.7%	0.0%
	future-genetic	2.9%	0.0%
	mnemonics	16.3%	0.0%
	par-mnemonics	18.7%	0.0%
	philosophers	15.3%	0.0%
	reactors	6.5%	0.1%
	rx-scrabble	8.7%	6.5%
	scala-doku	7.4%	0.2%
	scala-kmeans	0.6%	1.3%
	scala-stm-bench7	1.2%	0.0%
	scrabble	33.4%	0.7%
	Standard Overall	**9.3%**	**1.1%**

Suite	Benchmark	ER	COR
spring	helloworld	9.4%	0.0%
	petclinic:huge	19.0%	5.8%
	petclinic:large	11.3%	5.3%
	petclinic:medium	11.3%	5.7%
	petclinic:small	11.2%	5.1%
	petclinic:tiny	11.4%	4.8%
quarkus	helloworld	18.1%	0.0%
	tika:medium	5.2%	6.0%
	tika:small	5.2%	6.0%
	tika:tiny	5.2%	6.1%
micronaut	helloworld	13.8%	0.0%
	shopcart:huge	11.9%	0.0%
	shopcart:large	11.9%	0.0%
	shopcart:medium	11.9%	0.0%
	shopcart:small	11.9%	0.0%
	shopcart:tiny	11.9%	0.0%
	Mircroservice Overall	**11.3%**	**2.8%**

Exclusions

We prevent profiling and subsequent externalization of fields where we cannot safely adapt their accesses during compilation. We prevent externalization of fields of some internal types of Native Image, such as implementation classes of the garbage collector and the threading implementation. We also exclude types whose fields receive special treatment by the compiler, e.g., they are accessed unsafely via a known offset, or via `VarHandle` [31]. We further exclude volatile fields, as we currently cannot safely mimic their semantics after externalization.

Deallocation of Companion Objects

We designed our approach such that the allocation of a companion represents a one-way degradation. Hence, once a companion object for an object is allocated, there is no way to deallocate it, even if it is no longer needed at some later point (i.e., if all externalized fields in the companion object hold default values again). Only when the garbage collector collects the original object can the companion object also be collected. Chen et al. [9] solve this by identifying and collecting such "empty" companion objects during garbage collection.

However, this requires tighter integration with the garbage collector and also increases the duration of collection cycles. As companion objects should only be created for few objects, empty companions should appear even less frequently. We think optimizing for such a niche case is not necessary.

Deficiencies of the Heuristics

Our heuristics are based on observations on individual fields. While this enables a very precise calculation in terms of the saved memory per externalized field, it also limits the decision on whether or not to externalize at all to thresholds on individual fields. As an example, if a class has 5 fields for which profiling—based on our threshold of 5%—suggests externalization, then our approach would externalize them all. However, the profiling could be misleading here: In the best scenario for our approach, the sets of objects that have non-default values for the respective externalized field are perfectly overlapping. Then only a maximum of 5% of the objects would require a companion object. However, if these sets are completely disjoint, up to 25% of the objects could require a companion object. Identifying such cases via profiling would require profiling the composition of the field values *per object*. However, tracking all field combinations is infeasible due to the associated quadratic complexity.

7 Related Work

There is considerable related work on field and general memory footprint optimization techniques. However, most such works are implemented either on research compilers and VMs or in the context of a JIT compiler, which entails a different set of challenges but also opportunities compared to an AOT compiler. Nevertheless, they propose interesting and unique approaches that are in parts comparable to our work or propose other footprint optimization techniques using profiling information.

Chen et al. [9] implemented field externalization in the Kaffe VM [55], a Java Virtual Machine for embedded programs. They profile field usages and classify the fields into three levels based on their profiled value compositions: fields without a dominant value (level 0), fields with a dominant value other than the default value (level 1), and fields where the default value dominates (level 2). At run time, this information is picked up to, on the one hand, strip level 2 fields from objects in a similar manner to our approach and, on the other hand, *share* level 1 fields between objects. Similar to our approach, they use header bits to identify *compressed* (companion object not yet allocated) and *uncompressed*/*shared* objects (with allocated companion object/shared fields).

While they do not discuss their approach in the context of multithreading in detail, they seem to use some form of synchronization on accesses to externalized and shared fields. They note, however, that they can avoid the synchronization overhead in their target (embedded) JVMs, since they do not allow threads to preempt one another. Our approach is based on a standalone JVM, where multi-threading and concurrency are much more widespread. Thus, as detailed in Section 4.2.1, much of the effort of our approach went into designing semantically valid access patterns to externalized fields that conform to the Java memory model. This also complicates integrating their field sharing approach into our work.

Their approach is based on a JIT compiler and ours is based on an AOT compiler. Thus we also have different challenges: They use the interpreter to mark instructions that access externalized fields, which subsequently allows the JIT compiler to optimize them. Our approach is based on a closed-world assumption, thus we have information about all types that occur in the application. However, since there is no interpreter that can perform some levels of profiling when processing objects with externalized fields, all access variants have to be compiled immediately.

The results of our approach are hard to compare: They target embedded JVMs and, therefore, use a simulator and the SpecJVM98 benchmark suite [53] that has been retired since to gather benchmark results. We use a set of standard and microservice benchmarks to evaluate our approach.

Guo et al. [16] show an improvement to the aforementioned approach, again targeting an embedded JVM. They use the same field usage classification approach as Chen et al. [9] and generate meta-classes that describe the offsets of externalized and shared fields. As with Chen et al., the differences between our approach and theirs are that they target an embedded JVM, they scan the heap to gather profiling information, and the targeted benchmark set (SpecJVM98). Similarly, they do not mention their synchronization techniques to adhere to the Java memory model.

Ananian and Rinard [3] present a number of optimizations that also include field externalization to reduce the memory footprint of objects. This approach is implemented into a research Java AOT compiler (MIT FLEX compiler system). Their profiling approach is similar to ours; they introduce per-field counters in a separate profiling build and subsequently use a heuristic to select externalizable fields. They do not use companion objects that hold the externalized fields but use a hashtable that maps objects to their field values. The benefit of using a hashtable is that one single non-default-value write does not produce that much overhead compared to the allocation of the companion object. However, the hashtable itself, as well as the keys, require extra storage and introduce further indirections. Native Image supports lazily appending the identity hash code field and the monitor field to objects [37]. Introducing a hashtable for types with externalized fields would also require the identity hash code field for all types, thus reducing the savings potential.

Sartor et al. [47] present and summarize a number of techniques related to object compression. These also include the original approaches by Chen et al. [9] and Ananian and Rinard [3]. They implemented their approach on the Jikes RVM [2] but performed the evaluation without the optimizing compiler. Hence, comparing the results of our approaches is difficult, as we measured the real effects of field externalization after AOT compilation.

There is also work about other object compression techniques: Venstermans et al. [58] reduce or completely remove object headers in the Jikes RVM [2] by allocating all objects of a specific type in a contiguous memory region and using a *side array* for the header bits and status information used by the garbage collector. Chen et al. [8] implemented a custom garbage collector for the Sun KVM [54] that compresses objects and arrays when space is needed. However, every access to a compressed object subsequently has to decompress it first. Therefore, they also implemented a partitioning mechanism for objects and arrays that also allows lazy allocation of the individual partitions when needed. We see these approaches as orthogonal to our work on field externalization. Their work could allow us to compress objects even further, even after field externalization.

8 Conclusion

In this work, we presented a novel field externalization approach for modern Java VMs that reduces the footprint of objects by removing fields that mostly hold the default value. We use profiling information to identify such fields and subsequently move them into separate *companion classes*, which are generated at compile time. If, at run time, an externalized field is accessed, the corresponding *companion object* is allocated that holds the externalized fields.

The companion objects are referenced either via an additional injected field or a regular reference field that is reused as a *masked companion reference field*. Since our approach is integrated into a modern AOT compilation environment, GraalVM Native Image, we also introduce synchronization on accesses to externalized fields by using two header bits. An evaluation on a wide variety of benchmarks shows a modest but consistent reduction of the total allocated bytes across most benchmarks as well as a reduction in the max-RSS.

Overall, our work demonstrates the feasibility of field externalization in a state-of-the-art AOT compiler without sacrificing feature or language support but also highlights the challenges associated with it—particularly when adherence to the language semantics is of critical importance and when there is no run-time compilation or interpreter that enable fallbacks through recompilation.

References

1 Frances E. Allen and John Cocke. *A Catalogue of Optimizing Transformations*. Design and Optimization of Compilers. Prentice-Hall, 1972. Google-Books-ID: oeXaZwEACAAJ.

2 B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000. Conference Name: IBM Systems Journal. URL: https://ieeexplore.ieee.org/document/5387060, doi:10.1147/sj.391.0211.

3 C. Scott Ananian and Martin Rinard. Data size optimizations for java programs. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, LCTES '03, pages 59–68, New York, NY, USA, June 2003. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/780732.780741, doi:10.1145/780732.780741.

4 M. Arnold, S.J. Fink, D. Grove, M. Hind, and P.F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. *Proc. IEEE*, 93(2):449–466, February 2005. URL: http://ieeexplore.ieee.org/document/1386662/, doi:10.1109/JPROC.2004.840305.

5 Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: java benchmarking development and analysis. *SIGPLAN Not.*, 41(10):169–190, October 2006. URL: https://dl.acm.org/doi/10.1145/1167515.1167488, doi:10.1145/1167515.1167488.

6 Pohua P. Chang, Scott A. Mahlke, and Wen-Mei W. Hwu. Using profile information to assist classic code optimizations. *Software: Practice and Experience*, 21(12):1301–1321, 1991. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380211204. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380211204, doi:10.1002/spe.4380211204.

7 Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for FDO compilation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 42–52, Toronto Ontario Canada, April 2010. ACM. URL: https://dl.acm.org/doi/10.1145/1772954.1772963, doi:10.1145/1772954.1772963.

8 G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko. Heap compression for memory-constrained Java environments. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, pages 282–301, New York, NY, USA, October 2003. Association

for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/949305.949330, doi:10.1145/949305.949330.

9 Guangyu Chen, Mahmut Kandemir, and Mary J. Irwin. Exploiting frequent field values in java objects for reducing heap memory requirements. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, pages 68–78, New York, NY, USA, June 2005. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/1064979.1064990, doi:10.1145/1064979.1064990.

10 Jiho Choi, Thomas Shull, and Josep Torrellas. Reusable inline caching for JavaScript performance. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 889–901, New York, NY, USA, June 2019. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3314221.3314587, doi:10.1145/3314221.3314587.

11 Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. *SIGPLAN Not.*, 34(10):1–19, October 1999. URL: https://dl.acm.org/doi/10.1145/320385.320386, doi:10.1145/320385.320386.

12 L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 297–302, New York, NY, USA, January 1984. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/800017.800542, doi:10.1145/800017.800542.

13 Gilles Duboscq, Lukas Stadler, Thomas Wuerthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. Graal IR: An Extensible Declarative Intermediate Representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, page 9, Shenzhen, China, February 2013. URL: https://ssw.jku.at/General/Staff/GD/APPLC-2013-paper_12.pdf.

14 Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*, VMIL '13, pages 1–10, New York, NY, USA, October 2013. ACM. doi:10.1145/2542142.2542143.

15 Google. Profile-guided optimization - The Go Programming Language, 2024. URL: https://go.dev/doc/pgo.

16 Zhuang Guo, José Nelson Amaral, Duane Szafron, and Yang Wang. Utilizing field usage patterns for Java heap space optimization. In *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, CASCON '06, pages 6–es, USA, October 2006. IBM Corp. URL: https://dl.acm.org/doi/10.1145/1188966.1188974, doi:10.1145/1188966.1188974.

17 Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '01*, pages 54–61, Snowbird, Utah, United States, 2001. ACM Press. URL: http://portal.acm.org/citation.cfm?doid=379605.379665, doi:10.1145/379605.379665.

18 Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In Pierre America, editor, *ECOOP'91 European Conference on Object-Oriented Programming*, pages 21–38, Berlin, Heidelberg, 1991. Springer. doi:10.1007/BFb0057013.

19 Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. *SIGPLAN Not.*, 29(6):326–336, June 1994. URL: https://dl.acm.org/doi/10.1145/773473.178478, doi:10.1145/773473.178478.

20 Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture, Chapter 7 - Programming With General-Purpose Instructions. manual, Intel, October 2024. URL: https://www.intel.com/content/www/us/en/content-details/835781/

intel-64-and-ia-32-architectures-software-developer-s-manual-combined-volumes-1-2a-2b-2c-2d-3a-3b html.

21 Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference, A-Z, Chapter 4 Instruction Set Reference, M-U. manual, Intel, October 2024. URL: https://www.intel.com/content/www/us/en/content-details/835781/intel-64-and-ia-32-architectures-software-developer-s-manual-combined-volumes-1-2a-2b-2c-2d-3a-3b html.

22 Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a Java Just-In-Time compiler. *SIGPLAN Not.*, 35(10):294–310, October 2000. URL: https://dl.acm.org/doi/10.1145/354222.353191, doi:10.1145/354222.353191.

23 JavaNexus. Optimizing Java Performance with Thread-Local Allocation Buffers, May 2024. URL: https://javanexus.com/blog/java-performance-thread-local-allocation-buffers.

24 Roman Kennke, Vladimir Kozlov, Aleksey Shipilev, Erik Österlund, John Rose, Stefan Karlsson, and Thomas Stuefe. JEP 450: Compact Object Headers (Experimental), December 2024. URL: https://openjdk.org/jeps/450.

25 Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, page 75, USA, March 2004. IEEE Computer Society.

26 Junpyo Lee, Byung-Sun Yang, Suhyun Kim, Kemal Ebcioğlu, Erik Altman, Seungil Lee, Yoo C. Chung, Heungbok Lee, Je Hyung Lee, and Soo-Mook Moon. Reducing virtual call overheads in a Java VM just-in-time compiler. *SIGARCH Comput. Archit. News*, 28(1):21–33, March 2000. URL: https://dl.acm.org/doi/10.1145/346023.346037, doi:10.1145/346023.346037.

27 David Leopoldseder, Roland Schatz, Lukas Stadler, Manuel Rigger, Thomas Würthinger, and Hanspeter Mössenböck. Fast-path loop unrolling of non-counted loops to enable subsequent compiler optimizations. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ManLang '18, pages 1–13, New York, NY, USA, September 2018. Association for Computing Machinery. doi:10.1145/3237009.3237013.

28 David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. Dominance-based duplication simulation (DBDS): code duplication to enable compiler optimizations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization - CGO 2018*, pages 126–137, Vienna, Austria, 2018. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=3168811, doi:10.1145/3168811.

29 LLVM Project. How To Build Clang and LLVM with Profile-Guided Optimizations — LLVM 18.0.0git documentation, 2024. URL: https://llvm.org/docs/HowToBuildWithPGO.html.

30 Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 378–391, New York, NY, USA, January 2005. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/1040305.1040336, doi:10.1145/1040305.1040336.

31 Oracle. Java Development Kit Version 21 API Specification: VarHandle. URL: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/invoke/VarHandle.html.

32 Oracle. Java HotSpot Virtual Machine Performance Enhancements: Compressed Ordinary Object Pointer, September 2023. Publisher: September2023. URL: https://docs.oracle.com/en/java/javase/21/vm/java-hotspot-virtual-machine-performance-enhancements.html#GUID-932AD393-1C8C-4E50-8074-F81AD6FB2444.

33 Oracle. GraalPython, January 2025. original-date: 2018-04-17T09:54:59Z. URL: https://github.com/graalvm/graalpython.

34 Oracle. GraalVM, January 2025. URL: https://www.graalvm.org/.

35 Oracle. GraalVM JavaScript, January 2025. URL: https://github.com/oracle/graaljs.

36 Oracle. Native Image, January 2025. URL: https://www.graalvm.org/latest/reference-manual/native-image/.

37 Oracle. Object layout description in GraalVM Native Image when using Epsilon/Serial GC, January 2025. URL: https://github.com/oracle/graal/blob/261c38a7a47dd87f13ec3cbe0fbb85cfdff17963/substratevm/src/com.oracle.svm.hosted/src/com/oracle/svm/hosted/HostedConfiguration.java#L125-L145.

38 Oracle. TruffleRuby, January 2025. original-date: 2016-02-05T18:03:11Z. URL: https://github.com/oracle/truffleruby.

39 Karl Pettis and Robert C. Hansen. Profile guided code positioning. *SIGPLAN Not.*, 25(6):16–27, June 1990. URL: https://dl.acm.org/doi/10.1145/93548.93550, doi:10.1145/93548.93550.

40 Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. Renaissance: benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 31–47, New York, NY, USA, June 2019. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3314221.3314637, doi:10.1145/3314221.3314637.

41 Red Hat. Quarkus: Apache Tika Quickstart, March 2021. URL: https://github.com/quarkusio/quarkus-quickstarts/tree/758dd049407c6aa41c79c0d0dac1cff830ff9217/tika-quickstart.

42 Red Hat. Quarkus, December 2024. URL: https://quarkus.io/.

43 Manuel Rigger, Matthias Grimmer, and Hanspeter Mössenböck. Sulong - execution of LLVM-based languages on the JVM: position paper. In *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems - ICOOOLPS '16*, pages 1–4, Rome, Italy, 2016. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=3012408.3012416, doi:10.1145/3012408.3012416.

44 Graeme Rocher. Micronaut, December 2024. URL: https://micronaut.io/.

45 Nadav Rotem and Chris Cummins. Profile Guided Optimization without Profiles: A Machine Learning Approach, January 2022. arXiv:2112.14679 [cs]. URL: http://arxiv.org/abs/2112.14679, doi:10.48550/arXiv.2112.14679.

46 Barbara G. Ryder. Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Görel Hedin, editors, *Compiler Construction*, volume 2622, pages 126–137. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. Series Title: Lecture Notes in Computer Science. URL: http://link.springer.com/10.1007/3-540-36579-6_10, doi:10.1007/3-540-36579-6_10.

47 Jennifer B. Sartor, Martin Hirzel, and Kathryn S. McKinley. No bit left behind: the limits of heap data compression. In *Proceedings of the 7th international symposium on Memory management*, ISMM '08, pages 111–120, Tucson, AZ, USA, June 2008. ACM. URL: https://dl.acm.org/doi/10.1145/1375634.1375651, doi:10.1145/1375634.1375651.

48 Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da capo con scala: design and analysis of a scala benchmark suite for the java virtual machine. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 657–676, New York, NY, USA, October 2011. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/2048066.2048118, doi:10.1145/2048066.2048118.

49 Denys Shabalin. *Just-in-time performance without warm-up*. PhD thesis, EPFL, Lausanne, 2020. doi:10.5075/epfl-thesis-9768.

50 Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. Operating System Concepts, 10th Edition - Chapter 6: Synchronization Tools. In *Operating System Concepts, 10th Edition*, pages

257–288. Wiley, 10 edition, April 2018. URL: https://www.wiley.com/en-us/Operating+System+Concepts%2C+10th+Edition-p-9781119320913R150.

51 Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. Alias Analysis for Object-Oriented Programs. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Lecture Notes in Computer Science, pages 196–232. Springer, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-36946-9_8.

52 Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 165–174, Orlando, FL, USA, February 2014. ACM. doi:10.1145/2581122.2544157.

53 Standard Performance Evaluation Corporation. SPEC JVM98. URL: https://www.spec.org/osg/jvm98/.

54 Sun Microsystems, Inc. J2ME Building Blocks for Mobile Devices: White Paper on KVM and the Connected, Limited Device Configuration (CLDC). Technical report, Sun Microsystems, Inc., Palo Alto, CA, USA, May 2000. URL: https://www.oracle.com/technetwork/java/embedded/javame/java-mobile/kvmwp-150240.pdf.

55 Tim Wilkinson. The Kaffe Virtual Machine, December 2024. original-date: 2010-12-13T05:20:52Z. URL: https://github.com/kaffe/kaffe.

56 Po-An Tsai and Daniel Sanchez. Compress Objects, Not Cache Lines: An Object-Based Compressed Memory Hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 229–242, New York, NY, USA, April 2019. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3297858.3304006, doi:10.1145/3297858.3304006.

57 Tomoharu Ugawa, Stefan Marr, and Richard Jones. Profile Guided Offline Optimization of Hidden Class Graphs for JavaScript VMs in Embedded Systems. In *Proceedings of the 14th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, VMIL 2022, pages 25–35, New York, NY, USA, December 2022. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3563838.3567678, doi:10.1145/3563838.3567678.

58 Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. Java object header elimination for reduced memory consumption in 64-bit virtual machines. *ACM Trans. Archit. Code Optim.*, 4(3):17–es, September 2007. URL: https://dl.acm.org/doi/10.1145/1275937.1275941, doi:10.1145/1275937.1275941.

59 VMware. Spring, December 2024. URL: https://spring.io/.

60 VMware. Spring PetClinic Sample Application, December 2024. URL: https://github.com/spring-projects/spring-petclinic.

61 Christian Wimmer. GraalVM native image: large-scale static analysis for Java (keynote). In *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, VMIL 2021, page 3, New York, NY, USA, October 2021. Association for Computing Machinery. doi:10.1145/3486606.3488075.

62 Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. Initialize once, start fast: application initialization at build time. *Proc. ACM Program. Lang.*, 3(OOPSLA):184:1–184:29, October 2019. doi:10.1145/3360610.

63 Patrick Wintermeyer, Maria Apostolaki, Alexander Dietmüller, and Laurent Vanbever. P2GO: P4 Profile-Guided Optimizations. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, pages 146–152, New York, NY, USA, November 2020. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3422604.3425941, doi:10.1145/3422604.3425941.

64 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all.

In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, Onward! 2013, pages 187–204, Indianapolis, Indiana, USA, October 2013. ACM. doi:10.1145/2509578.2509581.

65 Pengfei Yuan, Yao Guo, and Xiangqun Chen. Experiences in profile-guided operating system kernel optimization. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, APSys '14, pages 1–6, New York, NY, USA, June 2014. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/2637166.2637227, doi:10.1145/2637166.2637227.